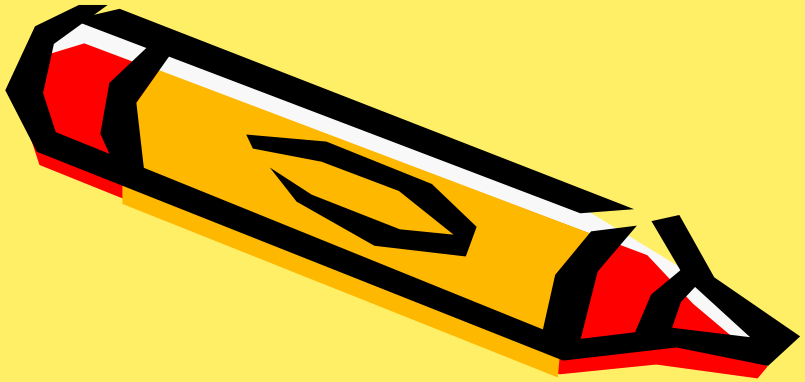
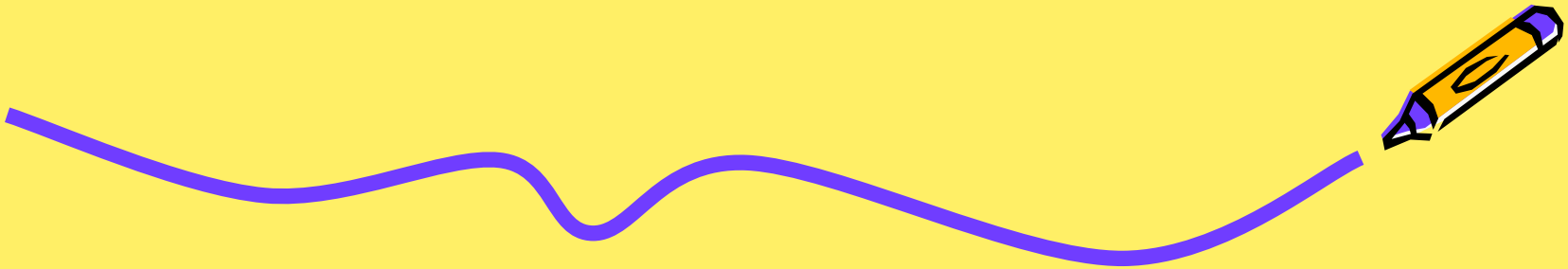


Chapter 4

Interprocess communication and remote invocation

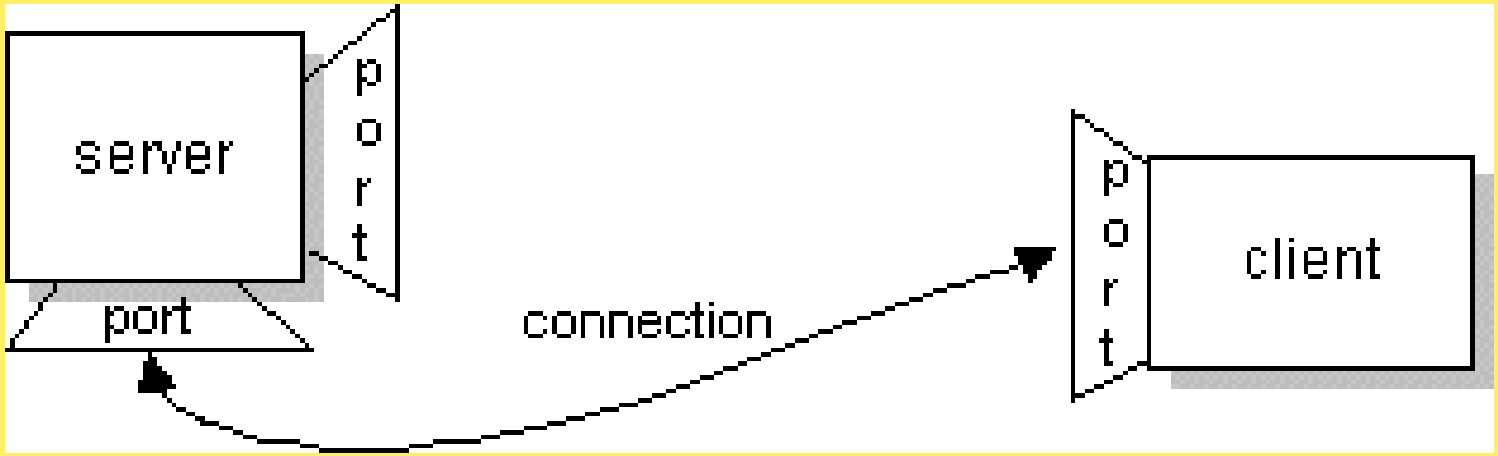


Interprocess communication

- In order for one process to communicate with another, one process sends a message to a destination and another process at the destination receives the message
- Send and receive operations are essential in the communication process
- Each process has an incoming messages queue
  - sending => means messages added to remote queues
  - receiving => messages removed from local queues
- Sending is non-blocking while receiving is blocking (synchronous) or non-blocking (asynchronous)
- Message is sent to (Internet address, local port) pairs that specifies another process.

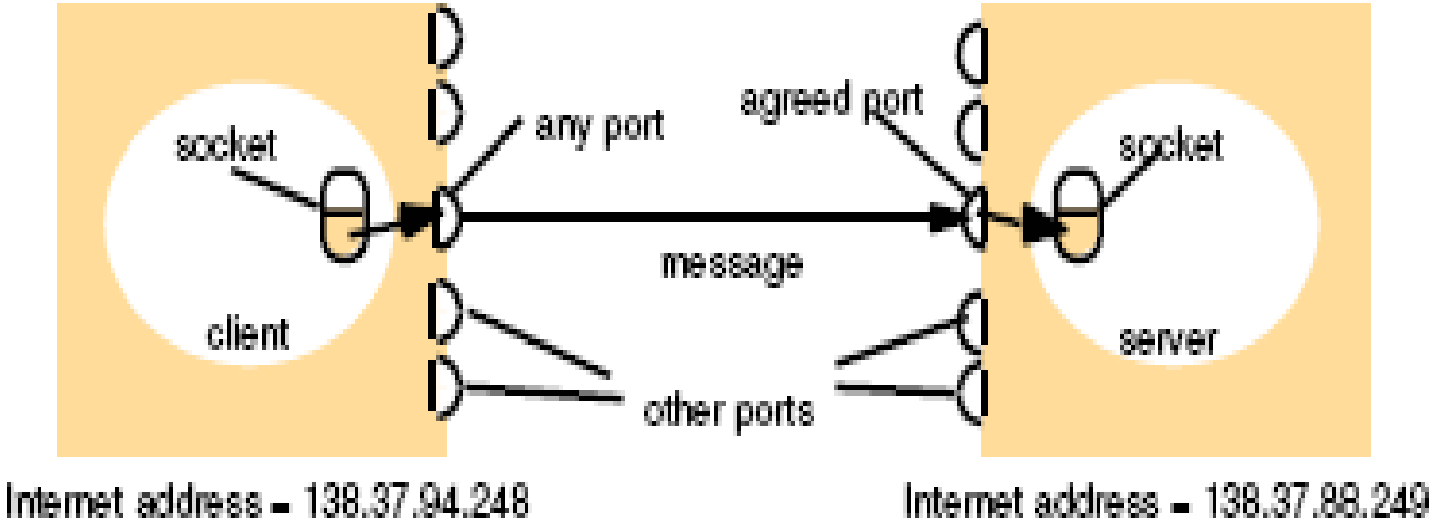
Ports

- Port numbers are numbers that are used for addressing messages to processes within a computer and are valid within this computer.
- A port number is a 16-bit integer
- Port define entry points for services provided by server applications.
- Important commercial applications such as Oracle have their own well known ports.
- A port has one receiver, many senders
- If a process has many senders, it may use multiple ports instead of just one to receive messages
- When an IP packet (i.e. message and its headers) is delivered to destination host, the TCP –or UDP- layer software directs it to the intended process via a specific port on that host.


server
port
port
connection
port
client

Sockets

- A socket is one of the most fundamental technologies of computer networking.
- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.
- A socket is used to allow one process to speak to another, very much like the telephone is used to allow one person to speak to another.
- A Socket must be bound (tied) to a local port and one of the Internet addresses of the computer on which it runs.
- Important: Any process may use multiple ports to receive messages but a process can not share ports with other processes on the same computer.

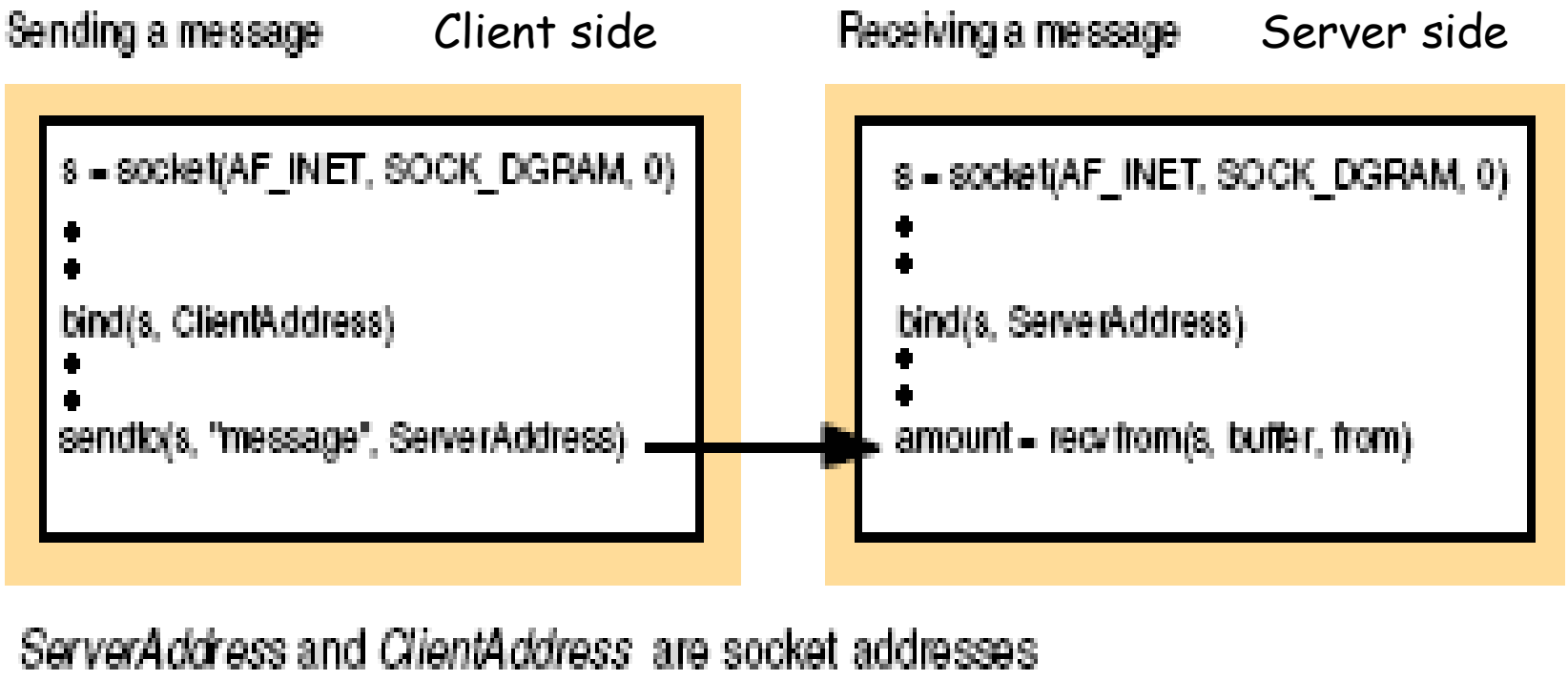


Relationship between messages, port and sockets

Sockets

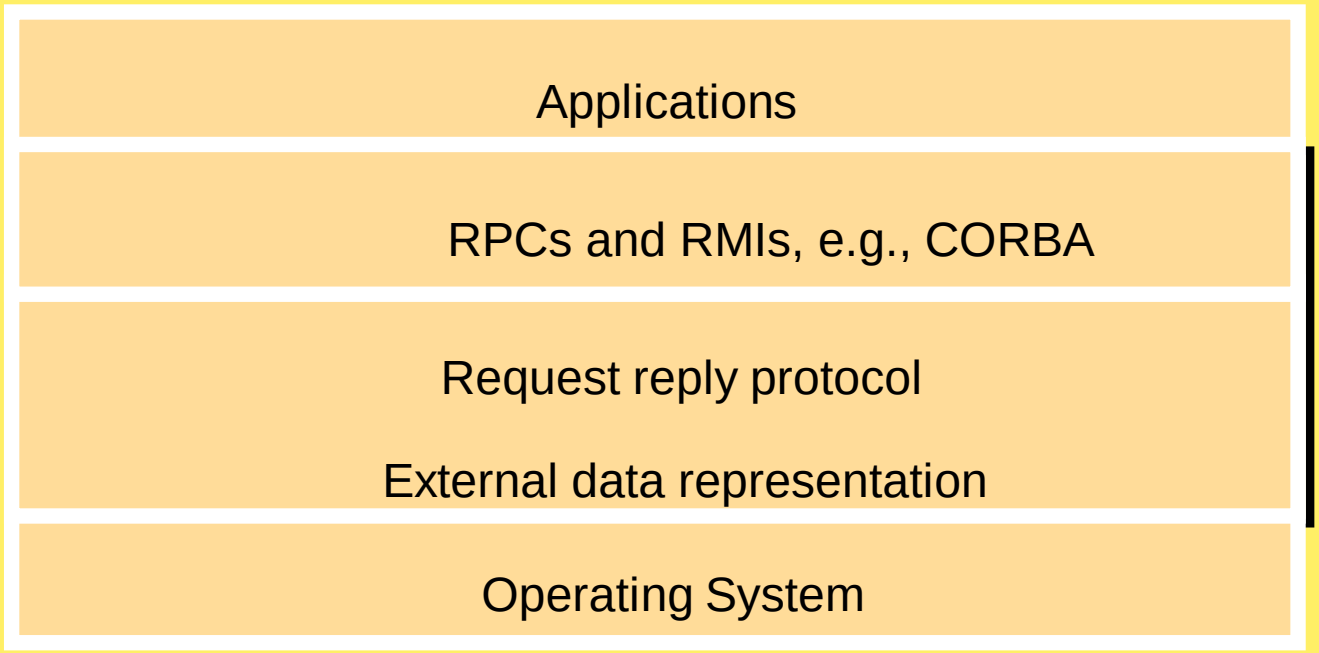
- A socket receives a message if it is was initiated to the port it is associated with and to this Internet address.
- Many processes can use the same socket for sending and receiving messages.
- Each computer has a large number of possible ports for use for local processes for receiving messages.
- Any number of processes may send messages to the same port which is connected to a socket which is the interface of a process.
- Each socket is associated with a particular protocol UDP or TCP.
- Many of today's most popular software packages -- including Web Browsers, Instant Messaging and File Sharing -- rely on sockets.
- Java provides a special class called InetAddress that represents Internet addresses

Example in Unix



- Socket must be bound to an address which is any available port number on server or client side (clientaddress and serveraddress)
- When sending, the sending (client) socket must know the port number of the receiving server.

Middleware for Interprocess communication



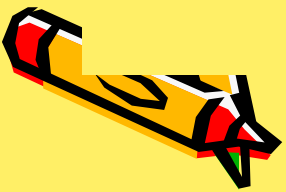
Middleware layers= *Provide support to the application*

- **When an application wishes to send a message to another application on another computer, the message is passed through the next layer (transport layer) through the middleware layers**

Middleware for Interprocess communication

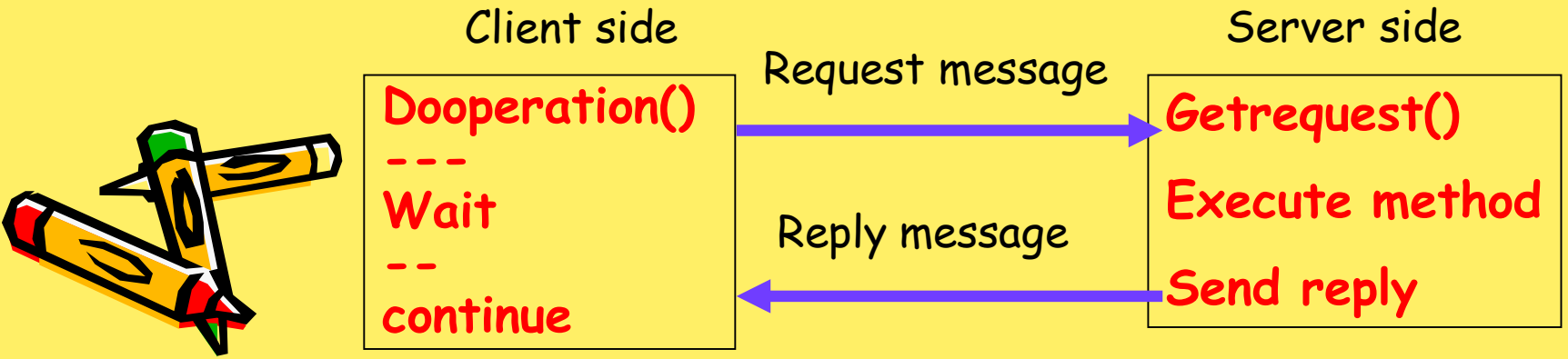
- A middleware is a group of programs that represent protocols based on message passing and processes basics to provide high-level abstraction to the users when communicating with each other.
- It is laid over the operating system to make it independent of the operating system type.
- The protocols that support the middleware are independent of the transport layer protocols (TCP and UDP).
- A protocol is a Formal set of rules that govern the *formats*, *contents*, and *meanings* of messages from computer to computer, process to process, etc.
- Must be agreed to by all parties to communicate.

- There are a group of protocols in this layer that we will study:
  - Request-reply protocol
  - External data representation and marshalling
  - Remote method invocation
  - Remote procedure invocation
  - Event-based invocation



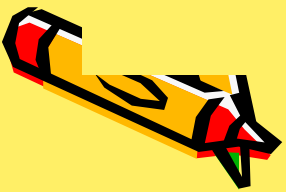
1- Request-reply protocol

- Used in client/server communication which is normally synchronous because the client process blocks until reply arrives from the server.
- It can be said that it is reliable because the reply is just like acknowledgment to the client.
- However, there is asynchronous request-reply communication in cases where clients can receive the reply later.
- Example of it is http protocol, which uses functions like: get, post, delete, trace,... to send and receive messages between server and client.



1- Request-reply protocol

- If used in UDP datagram protocol, there is no guarantee that the sending of a request message will result in a method being executed.
- The system should keep history of the previously executed messages to avoid to re-execute the method in the request when transmitting reply messages.



2- Data representation and marshalling

- We must note that the information stored in running programs is represented as data structures whereas the information in messages consists of sequence of bytes.
- Data structures must be flattened (converted to a sequence of bytes) before transmission and then rebuilt on their arrival to the other side.
- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- It translates structured data and primitive values into data representation for transmission.
- UnMarshalling is the process of generating of primitive values from data representation and building data structures.

Example of Marshalling

- **CORBA's common data representation:**
- Has 15 primitive types of data
- the type of data item is not given with the data representation in the message
- assumed that sender and recipient have common knowledge of the order and types of the data items in a message
- marshalling operations generated automatically from the specification of datatypes
- CORBA interface compiler generates marshalling and unmarshalling operations

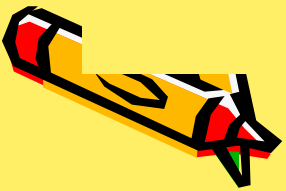
XML (Extensible Markup language) as an example of marshalling

- XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services.
- XML was originally made for WWW but then was used in archiving and retrieval systems as it is readable on any computer.
- XML data items are tagged with makeup strings.
- The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures.
- Example of XML:

```
<person id = "1234">
    <name> amany </name>
    <place> domyat </place>
    <year> 1967 </year>
</person>
```

XML as example of marshalling

- The tags are related to the text they enclose unlike in HTML where tags specify how the browser display the text.
- XML parsers and generators are available for most commonly used programming languages.
- For example, Java software for writing out Java objects as XML (marshalling) and for creating Java object from such structures (unmarshalling)



Distributed objects and remote invocations

- Applications that are composed of cooperating programs running in several different processes.
- Such programs need to invoke operations in other processes running on different computer.

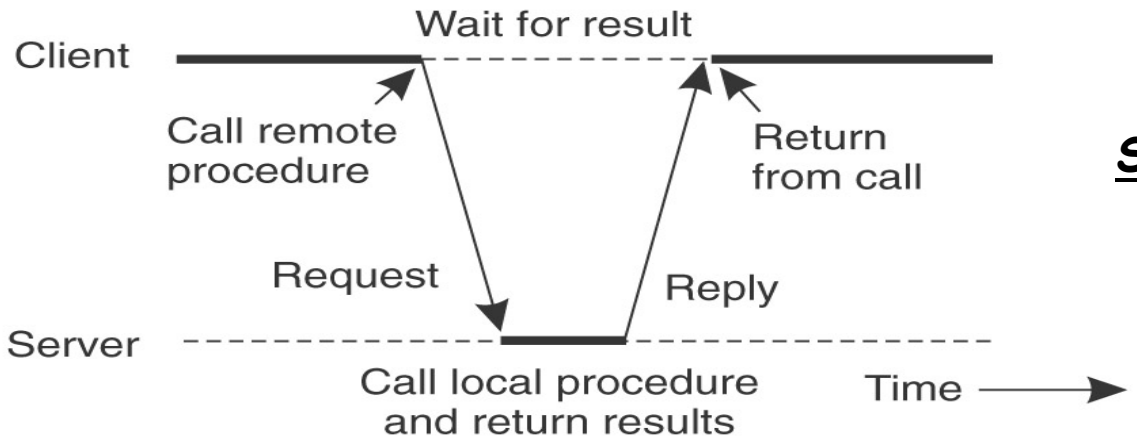
Programming models:

1. RPC (Remote Procedure Call)

- Remote Procedure Call allows a client to execute procedures on other computers.
- Client programs call procedures in server program running in separate process in different computer.
- *The* most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
  - NFS (Network File System) is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems

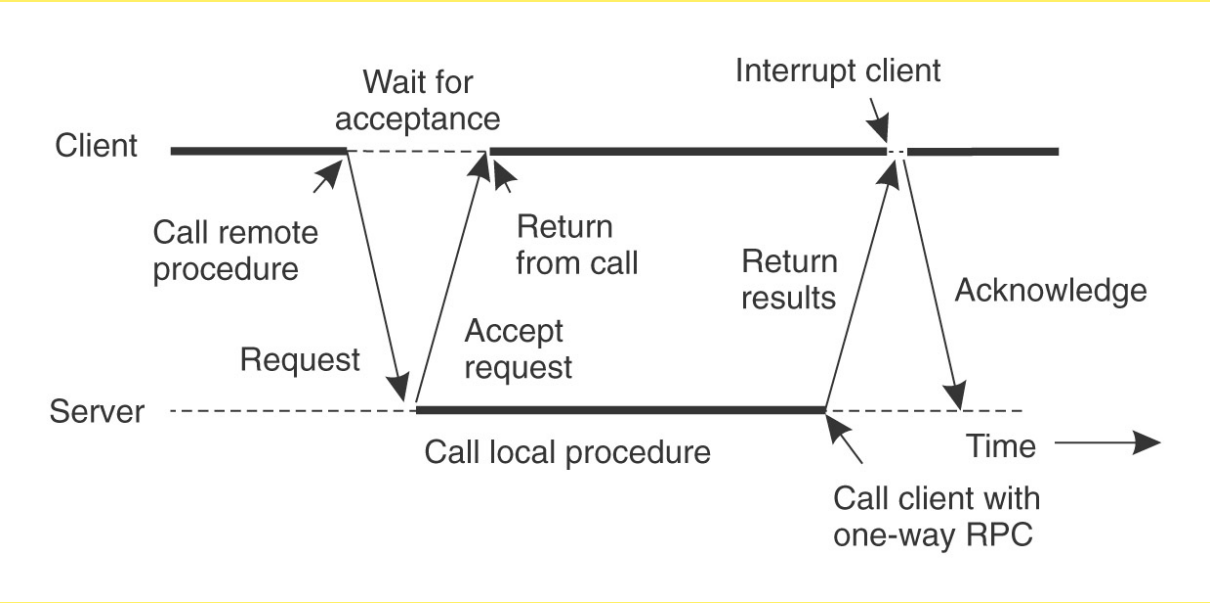
1. RPC (Remote Procedure Call)

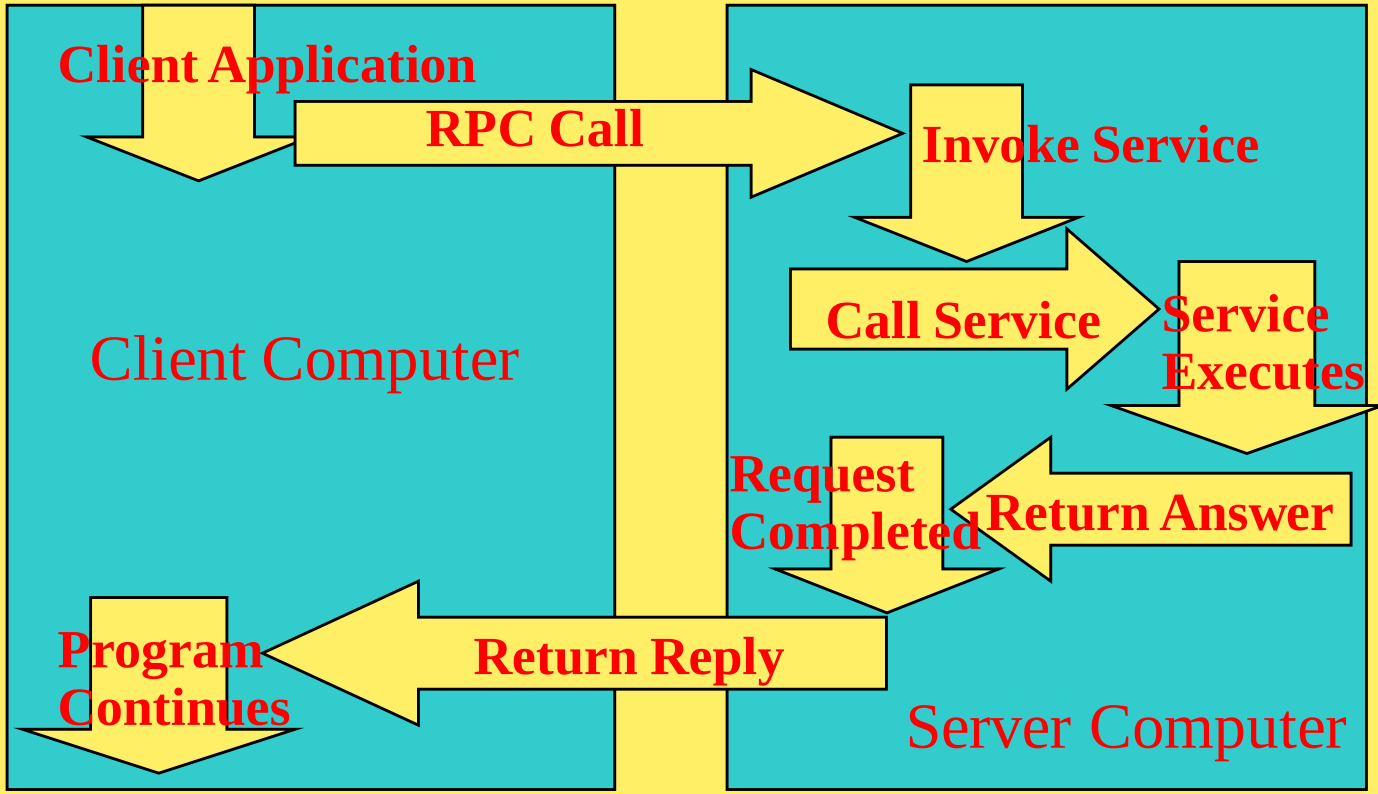
- **Fundamental idea: –**
  - Server process exports an *interface* of procedures or functions that can be called by client programs
    - similar to library API, class definitions, etc.
- Clients make remote procedure/function calls
  - *As if* directly linked with the server process
  - Procedure/function call is converted into a message exchange with remote server process



Synchronous RPC

Asynchronous RPC

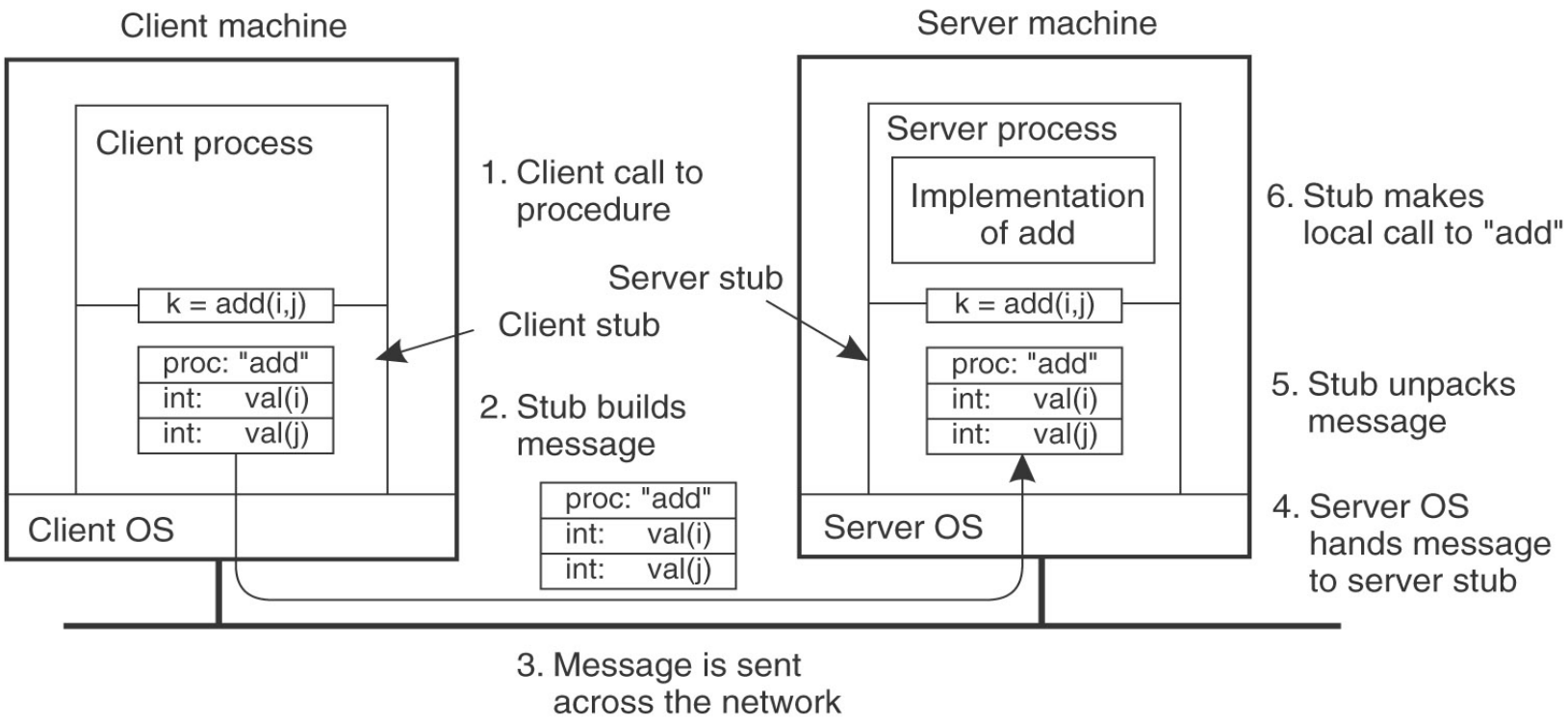




- RPC simplifies the task of writing client/server programs.
- An RPC is really a *process invocation*, not a *procedure call*.
- An invoked program runs in another domain.
- RPCs hide the complexity of the network by using the procedure call mechanism familiar to programmers.
- They make life easy for programmers, but hard for Network OS designers.

Using stubs to implement RPC

- Implementation — to make a pair of *Stubs* (also known as a proxy)
- *A client-side stub* is a function that looks to the client as if it were a callable function of the service.
- *A service-side stub* looks like a client calling the service
- The client program thinks it's invoking the service but it's calling the client-side stub
- The service program thinks it's called by the client but it's really called by the service-side stub
- The stubs send messages to each other to make the RPC happen transparently.


Client machine
Server machine
Client process
Server process
Implementation of add
k = add(i,j)
k = add(i,j)
proc: "add"
int: val(i)
int: val(j)
proc: "add"
int: val(i)
int: val(j)
1. Client call to procedure
Server stub
Client stub
2. Stub builds message
proc: "add"
int: val(i)
int: val(j)
6. Stub makes local call to "add"
5. Stub unpacks message
4. Server OS hands message to server stub
Client OS
Server OS
3. Message is sent across the network

RPC Stubs – Summary

- *Client-side stub*
  - Looks like local server function to the client
  - Same interface as local function
  - Bundles arguments into a message, sends to server-side stub
  - Waits for reply, unbundles results
  - returns

- *Server-side stub*
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

RPC – Issues

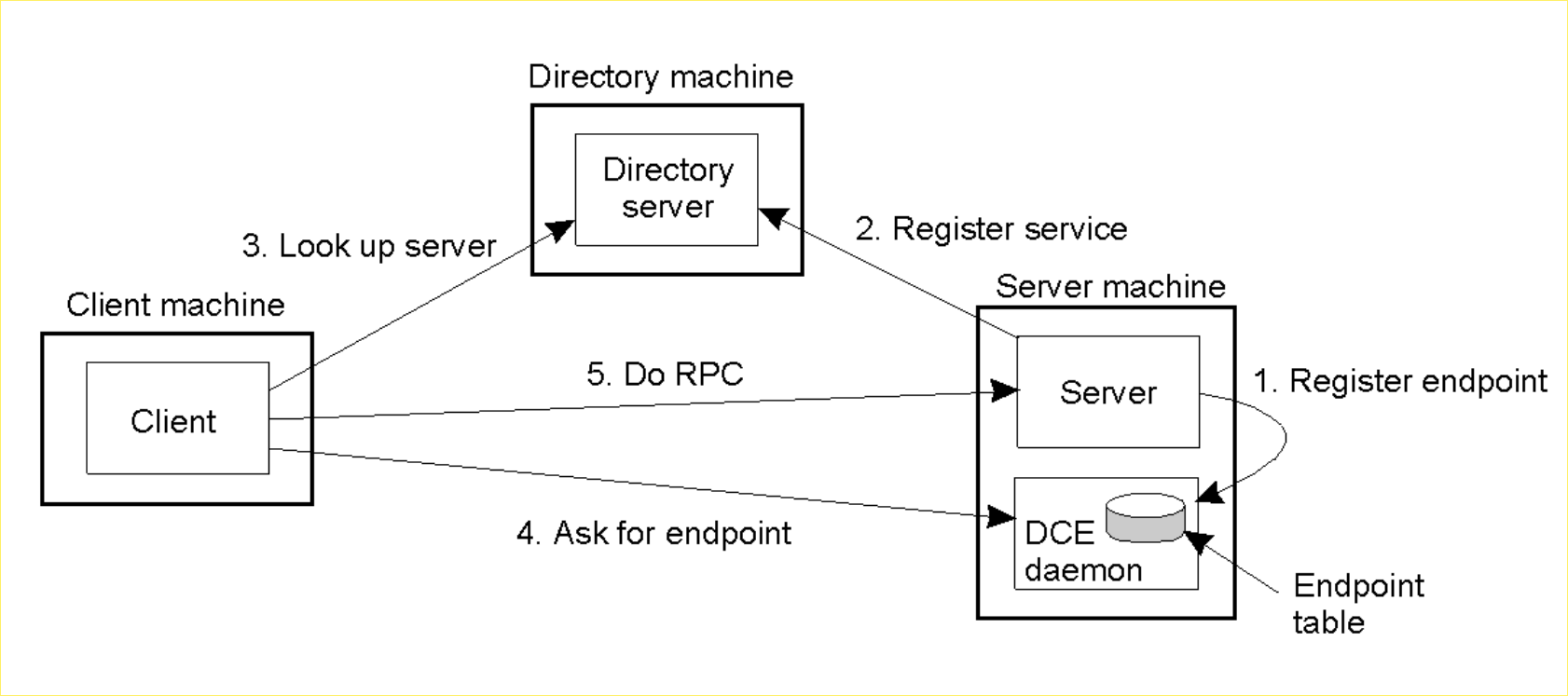
- How to make the "remote" part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- Solutions:
- A server defines the service interface using an *interface definition language* (IDL)
- The IDL specifies the names, parameters, and types for all client-callable server functions
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
  - *Server-side* and *client-side*
- IDL must also define representation of data on network
  - Multi-byte integers
  - Strings, character codes
  - Floating point, complex, ...
  - ...

- **Linking:–**
  - Server programmer implements the service's functions and links with the *server-side* stubs
  - Client programmer implements the client program and links it with *client-side* stubs
- **Operation:–**
  - Stubs manage *all* of the details of remote communication between client and server
- *Marshalling*
  - the packing of function parameters into a message
- *Unmarshalling*
  - the extraction of parameters from a message
- Function call:–
  - Client stub marshals the arguments into message
  - Server stub unmarshals the arguments and uses them to invoke the service function
- Function return:–
  - Server stub marshals return values into message
  - Client stub unmarshals return values and returns them as results to client program

RPC Binding

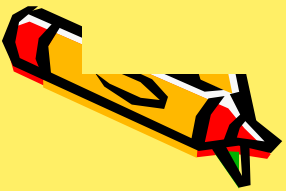
- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Client-to-server binding in DCE.



Remote Procedure Call is used ...

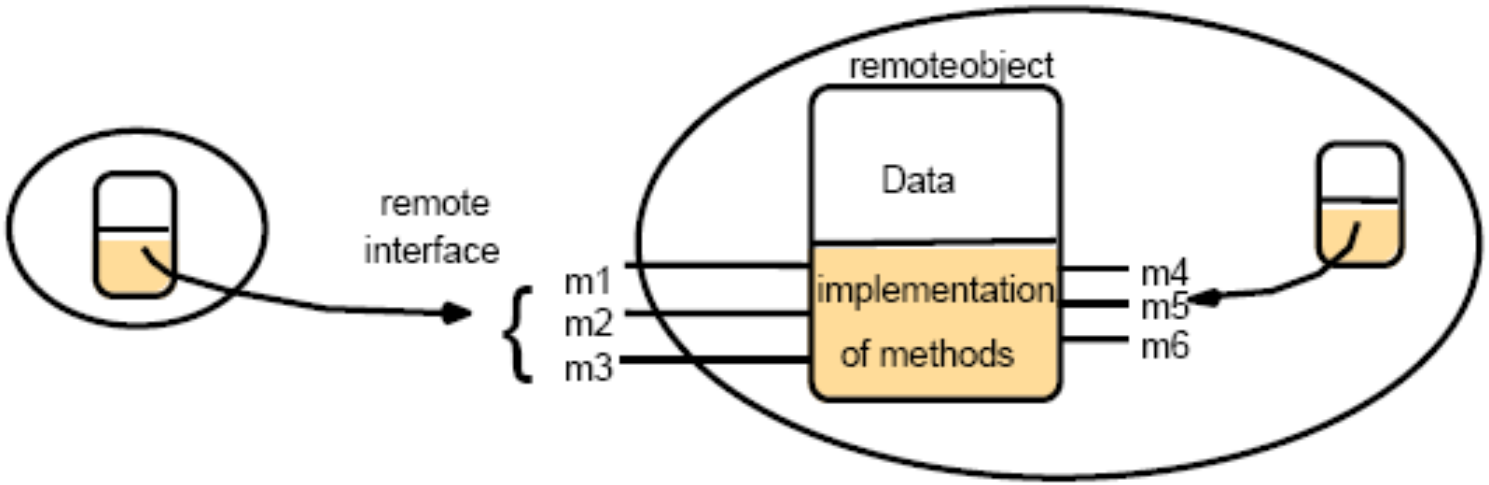
- Between processes on different machines
  - E.g., client-server model
- Between processes on the same machine
  - More structured than simple message passing
- Between subsystems of an operating system
  - Windows XP (called *Local Procedure Call*)

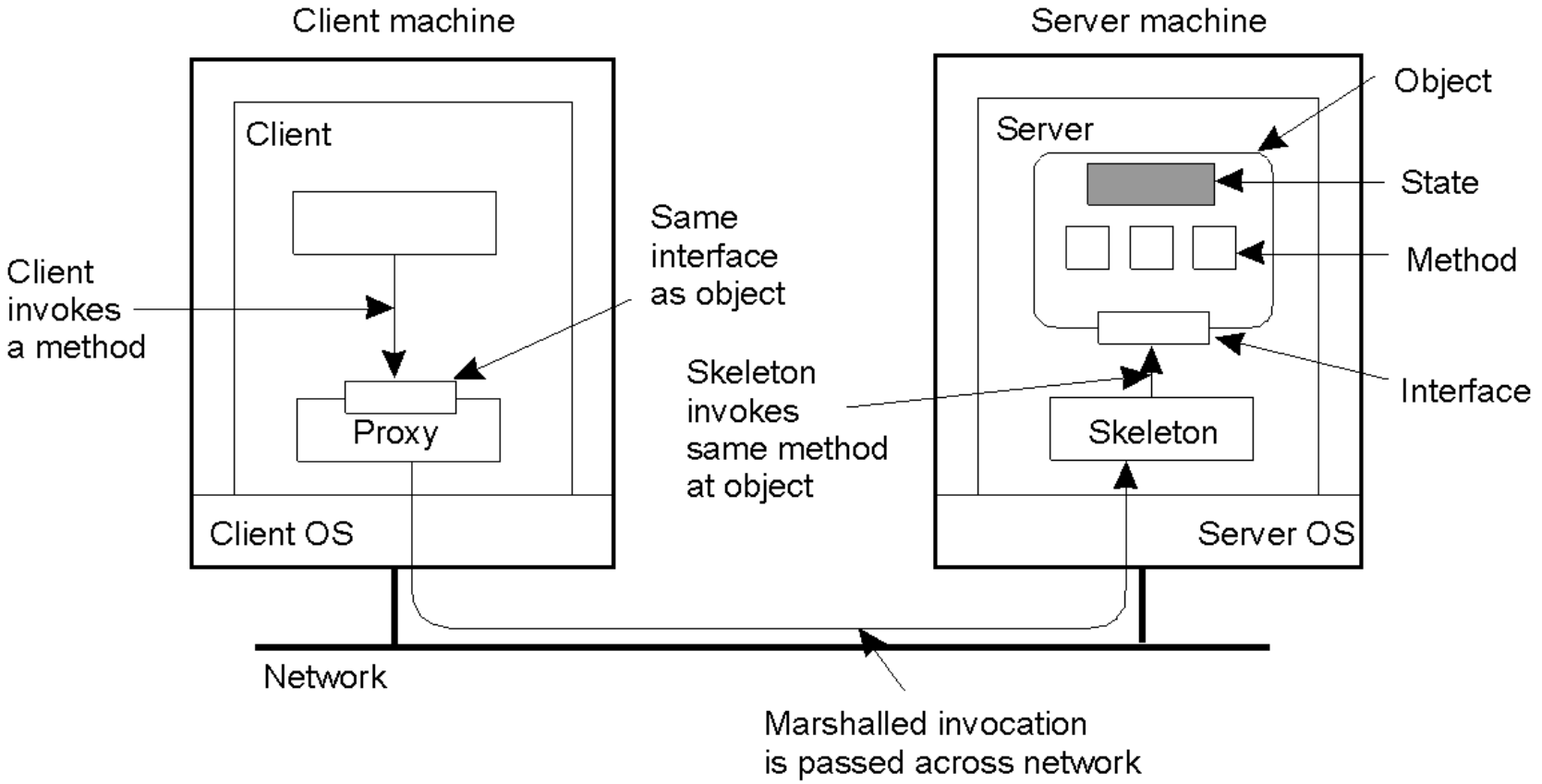


2. RMI (Remote Method Invocation)

- Appeared Due to the appearance of object-oriented programming.
- Objects in different processes communicating with one another by invoking methods of an object.
- It is an extension of RPC.
- Remote objects are those objects that can receive remote invocations.
- A method in the remote interface of an object is invoked synchronously, with the invoker waiting for reply
- Remote object reference
  - an identifier that can be used globally *throughout a distributed system* to refer to a particular unique remote object.
- Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

A remote object and its remote interface




Client machine
Server machine
Client
Server
Object
State
Method
Interface
Client invokes a method
Same interface as object
Skeleton invokes same method at object
Proxy
Skeleton
Client OS
Server OS
Network
Marshalled invocation is passed across network

Programming models

3. Event-based programming

- Allow objects to receive notification of the events of other objects in which they have registered interest.
- Notifications are sent asynchronously to multiple subscribers when a published event occurs at an object of interest

- **Modules and objects Interface**
- A program is divided into a set of modules that can communicate with one another by procedure call between modules or by directly access the variables in another module.
- Each module has interface to control the interaction between modules.
- This interface specifies the procedures and the variables that can be accessed by other modules (for example, public attributes and methods).

Modules and objects Interface

- The internal part of the module is hidden to the outside, only they can see the interface.
- Thus, the internal implementation of a module can be changed without affecting the other communicating modules.
- **Interface in distributed systems**
- Modules can run in separate processes in different computers.
- This introduces many problems as follows:
- A module running in one process can not access the variables in a module in another process.
- Thus, a module that will use RPC or RMI can not specify direct access to variables.
- Also, the call by value and call by reference mechanisms for parameter passing can not also be used when the caller and the procedure are in different processes.

Interface in distributed systems

- Pointers in one process are not valid in another remote one.
- These problems were solved in the client server model as we will see.
- Each server provides a set of procedures that are available for use by the clients.
- For example, a file server provides procedures for reading and writing files.
- The service interface refers to the procedures offered by a server defining the types of inputs, outputs arguments of each procedure.
- A procedure in RPC (or a method in RMI) in the interface of a module in distributed program describes the parameters as input, outut or both.

Client-server solution to interface

- Input parameters are passed to the remote module by sending the values of the arguments in the request message and then supply them as arguments to the operation to be executed in the server.
- Output parameters are returned in the reply message and are used as the result of the caller to replace the values of the corresponding variables in the calling evironment.
- When used for both, the value must be translated in both the request and reply messages.

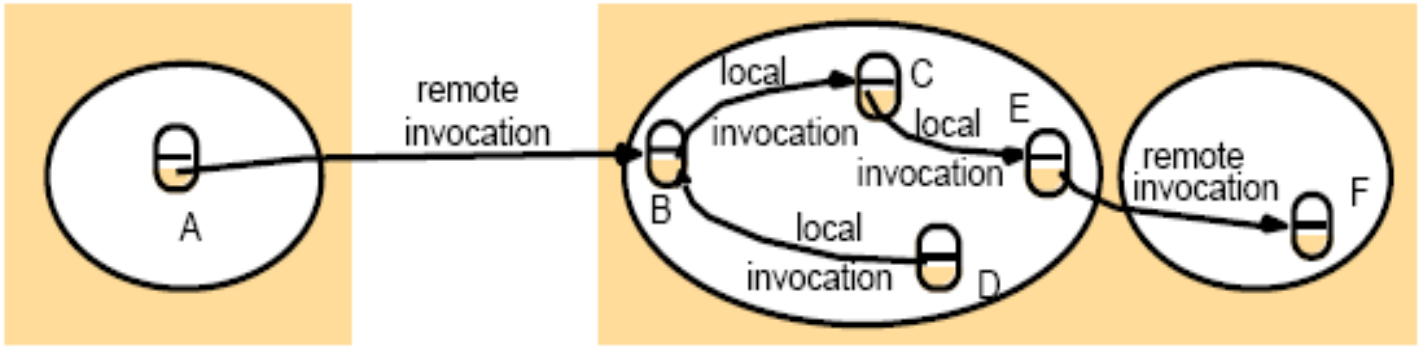
Remote Interface

- In distributed object model, remote interface specifies the methods of an object that are available for invocation by objects in other processes, defining the types of inputs, outputs arguments of each of them.
- The difference between procedures and methods is that the methods in remote interfaces can pass objects as arguments and results of methods.
- Also, reference to remote objects may be passed (they are different from pointers).
- To solve the pointers problem we have two options:
- Option 1: use *call by value*
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub builds up a new pointer
- Option2: use *call by result*
  - Sending stub provides buffer, called function puts data into it

  - Receiving stub copies data to caller's buffer as specified by pointer
- Option3: use *call by value-result*
  - Caller's stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply

- Not allowed:–
  - *Call by reference*
  - *Aliased arguments*

Remote and local method invocations



Local invocation=between objects on same process.

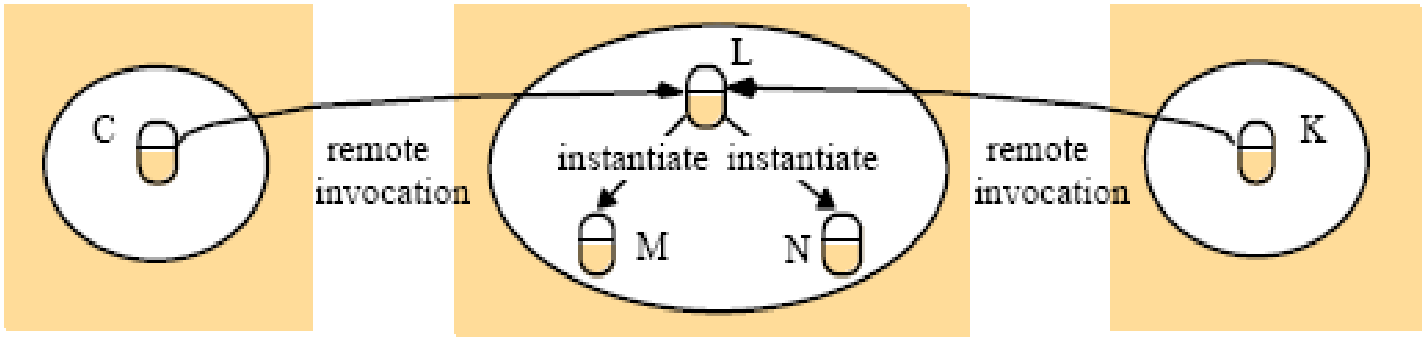
- Has *exactly one* semantics

Remote invocation=between objects on different processes.

- Ideally also want *exactly one* semantics for remote invocations

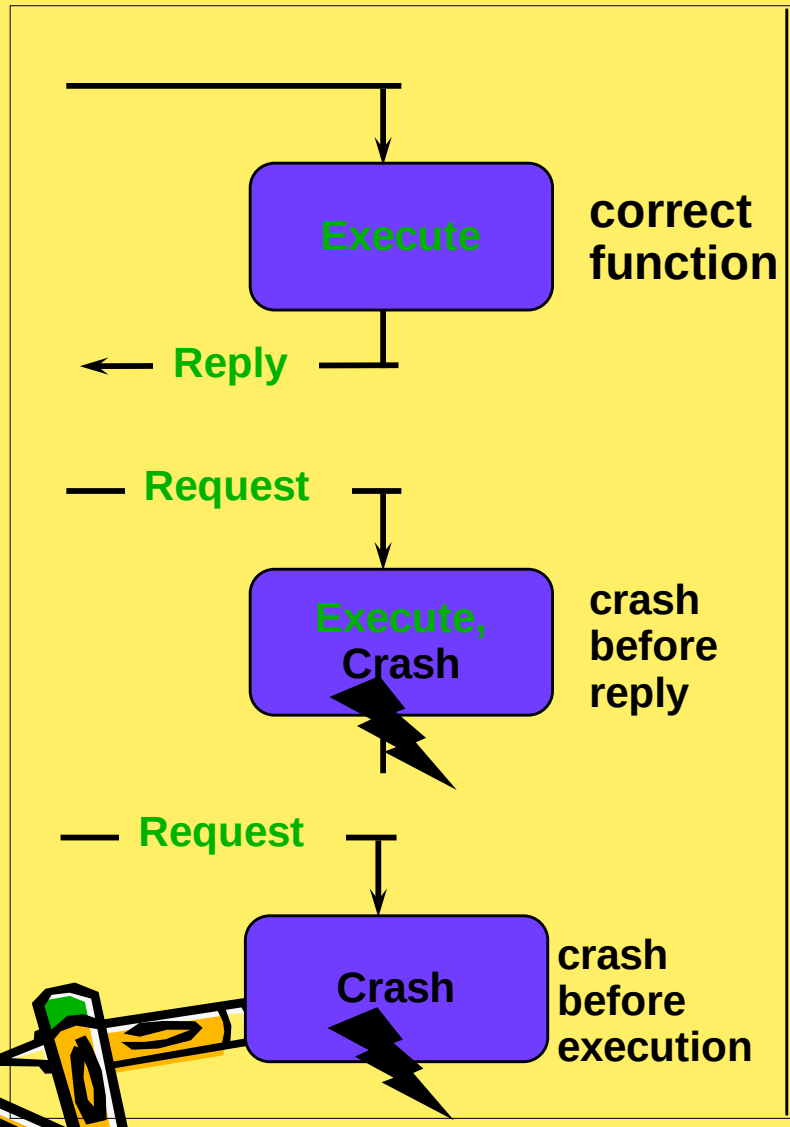
But difficult (why?)

Figure 5.5 Instantiation of remote objects



Failure Modes of RMI/RPC

(and if request is received more than once?)

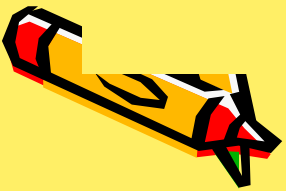


TCP and UDP

- TCP (*Transport control protocol*)
  - *Connection*-based, which means that the sender and receiver must cooperate to establish a bi-directional connection.
  - It is a reliable transmission as it uses acknowledgement to ensure the receiving of the packet.
  - It presents additional mechanisms to meet reliability guarantee like sequencing, flow control, retransmission,...
- UDP (User datagram protocol)
  - Unreliable transmission
  - A UDP datagram is encapsulated in an IP packet.
  - It has shorter header (source port number, destination port number, length and checksum).
  - Does not use acknowledgement, so there is no guarantee of packet delivery.

TCP and UDP

- It has minimum cost, minimum transaction delays, no setup costs, no administrative acknowledgment message.
- It does not provide reliability mechanisms except checksum which is optional.
- Used in unreabile applications or services that require no reliable delivery of single or multiple messages.



How RPC Systems Work

- When the server starts, it registers itself with the portmapper.
- It also registers the RPC program numbers, and versions.
- Before the client can make an RPC call to the server, it consults the portmapper of the server to identify the port number.
- The client and server open a communication path to execute remote procedures.
- Example: This example program will shows you how to use a server program, a client program and an interface definition file to let the client program call the functions in the server program and get the results.
- The client will send an integer to server; the server will calculate its square and then send the square value back to the client.

RPC Example (.x file)

- The Interface Definition file: square.x

```
struct square_in { /* input argument */
   long arg1;
};
struct square_out { /* output result */
   long res1;
};
program SQUARE_PROG {
    version SQUARE_VERS {
        square_out SQUAREPROC(square_in) = 1;
    } = 1;
} = 0x31234567;
```

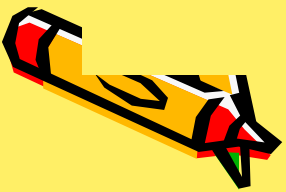
RPC Example (rpcgen output)

```
goode-8 RPC>: rpcgen -N -a square.x
goode-9 RPC>: ls -al
total 34
```

RPC Example (header file-1)

```
#ifndef _SQUARE_H_RPCGEN
#define    _SQUARE_H_RPCGEN

#include <rpc/rpc.h>
struct square_in {
  long arg1;
};
typedef struct square_in square_in;
struct square_out {
  long res1;
};
```



RPC Example (header file-2)

```
typedef struct square_out square_out;

#define    SQUARE_PROG      0x31234567
#define    SQUARE_VERS      1
#define    SQUAREPROC 1

extern  square_out * squareproc_1();
extern int square_prog_1_freeresult();

/* the xdr functions */
extern bool_t xdr_square_in();
extern bool_t xdr_square_out();
#endif /* !_SQUARE_H_RPCGEN */
```

RPC Example (square_xdr.c-1)

```
#include "square.h"

bool_t
xdr_square_in(xdrs, objp)
    register XDR *xdrs;
    square_in *objp;
{

#if defined(_LP64) || defined(_KERNEL)
    register int *buf;
#else
    register long *buf;
#endif
```

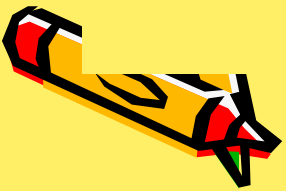
RPC Example (square_xdr.c-2)

```
if (!xdr_long(xdrs, &objp->arg1))
          return (FALSE);
    return (TRUE);
}

bool_t
xdr_square_out(xdrs, objp)
    register XDR *xdrs;
    square_out *objp;
{
```

```
#if defined(_LP64) || defined(_KERNEL)
  register int *buf;
#else
  register long *buf;
#endif

  if (!xdr_long(xdrs, &objp->res1))
      return (FALSE);
  return (TRUE);
}
```



RPC Example (square_clnt.c-1)

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "square.h"
#ifndef _KERNEL
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#endif /* !_KERNEL */

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };
square_out *
squareproc_1(arg1, clnt)
    square_in arg1;
    CLIENT *clnt;
{
    static square_out clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, SQUAREPROC,
           (xdrproc_t) xdr_square_in, (caddr_t) &arg1,
           (xdrproc_t) xdr_square_out, (caddr_t) &clnt_res,
           TIMEOUT) != RPC_SUCCESS) {
           return (NULL);
    }
    return (&clnt_res);
}
```